Integrating a Tiny Angular Front-End with Your FastAPI Spam Classifier

Below is a concise, step-by-step guide that starts from an **existing, running** FastAPI deployment on Render and ends with a **single-page Angular UI** that calls /predict and shows "Spam" or "Ham".
The flow keeps your current backend untouched; you will only add CORS configuration and create a small Angular project that can be hosted either (a) as static files on Render **or** (b) from any static host (e.g., GitHub Pages, Netlify, Vercel, S3).

1. Backend tweaks (CORS only)

1. Open app/api.py
2. Install and enable CORS middleware:

```
pip install fastapi[all]    # you already have FastAPI, but ensure 'fastapi[all]' for COR
```

```
from fastapi.middleware.cors import CORSMiddleware

app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],          # lock this down later if needed
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
```

3. Commit & push. Render will redeploy automatically.

2. Generate a minimal Angular app

1. Install Angular CLI (if not already):

```
npm install -g @angular/cli
```

2. Create a project in a sibling folder to spam-classifier-api:

```
ng new spam-ui --minimal --routing=false --style=css
cd spam-ui
```

The `--minimal` flag avoids testing boilerplate and extra files.

3. Build the UI

3.1 Template (src/app/app.component.html)

```
<div class="container">
  <h1>Spam Classifier</h1>

  <textarea
    [(ngModel)]="message"
    placeholder="Type or paste text..."
    rows="6"
  ></textarea>

  <button (click)="classify()">Classify</button>

  <p *ngIf="result !== null">
    Result: <strong>{{ result }}</strong>
  </p>
</div>
```

3.2 Component logic (src/app/app.component.ts)

```
import { Component } from '@angular/core';
import { HttpClient } from '@angular/common/http';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
})
export class AppComponent {
  message = '';
  result: string | null = null;

  constructor(private http: HttpClient) {}

  classify() {
    if (!this.message.trim()) { return; }
    this.http
      .post<any>('https://spam-classifier-api-vrk2.onrender.com/predict', {
        text: this.message,
      })
      .subscribe(
        (resp) => (this.result = resp.prediction ? 'Spam' : 'Ham'),
        (err) => (this.result = 'Error')
      );
  }
}
```

3.3 Module setup (src/app/app.module.ts)

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { HttpClientModule } from '@angular/common/http';

import { AppComponent } from './app.component';

@NgModule({
  declarations: [AppComponent],
  imports: [BrowserModule, FormsModule, HttpClientModule],
  bootstrap: [AppComponent],
})
export class AppModule {}
```

3.4 Basic styling (src/styles.css)

```
body { font-family: sans-serif; background:#fafafa; }
.container { max-width:480px; margin:40px auto; padding:20px; background:#fff; border:1p
textarea { width:100%; font-size:1rem; }
button { margin-top:10px; padding:8px 16px; }
```

4. Local test

```
ng serve
```

Open http://localhost:4200, type text, click **Classify**, confirm result.

5. Production build

```
ng build --configuration=production
```

Angular outputs static files to `dist/spam-ui/`.

6. Deployment options

Option A – Serve UI from the same Render service (fastest)

1. In spam-classifier-api repository, create `app/static` folder.
2. Copy everything from `dist/spam-ui/` into `app/static/`.
3. Add a route in FastAPI to serve the Angular index:

```
from fastapi.staticfiles import StaticFiles
from fastapi.responses import FileResponse
import pathlib

app.mount("/static", StaticFiles(directory="app/static"), name="static")

@app.get("/", include_in_schema=False)
def ui():
    index_path = pathlib.Path("app/static/index.html")
    return FileResponse(index_path)
```

4. Commit, push → Render redeploys.
5. Visiting your Render URL root (e.g., https://spam-classifier-api-vrk2.onrender.com/) now shows the UI; API remains at /predict.

Option B – Deploy UI as a separate static site

1. Push `dist/spam-ui/` to GitHub Pages, Netlify, Vercel, or S3.
2. No backend change except CORS (already done).
3. Update API URL in component if deploying under a custom domain.

7. Automation (optional)

Add an npm script to build and copy UI into backend automatically:

```
// package.json in spam-ui
"scripts": {
  ...
  "deploy-backend": "ng build --configuration=production && cp -r dist/spam-ui/* ../spam-
}
```

Run `npm run deploy-backend`, then push backend repo.

8. Security & polish

- Replace `"*"` in `allow_origins` with your final UI domain.
- Add simple input length validation on the Angular side.
- Show a spinner during the HTTP call for better UX.
- Consider HTTPS-only cookies or auth headers if the API evolves.

9. Project layout after integration

```
spam-classifier-api/
│
├─ app/
│   ├─ api.py
│   ├─ model.joblib
│   ├─ static/        ← Angular build output
│   └─ ...
├─ spam-ui/          ← Angular source (optional, can live in separate repo)
└─ ...
```

You now have a lightweight Angular front-end seamlessly calling your FastAPI spam classifier and can extend both halves independently.

⁂

[^1_1]: https://spam-classifier-api-vrk2.onrender.com/predict. --- # When I create app using ng new spam-ui --minimal --routing=false --style=css, then there is no app.component.html within src/app/, there app.ts and app.config.ts are there The Angular CLI changed its **default project layout** in v17 (2023-11) when it introduced the *stand-alone components* model. With `ng new … --minimal`, you now get: ``` src/app/ │ ├─ app.component.ts ← component class **with** inline template & styles ├─ app.config.ts ← providesRouter(), environment, etc. └─ main.ts ``` Because the template is *inlined*, the separate `app.component.html` and `app.component.css` files are no longer generated. ### What to do 1. **Keep the inline template** (simplest) Open `app.component.ts` and insert your HTML directly in the `@Component` decorator's `template` property: ```ts @Component({ selector: 'app-root', standalone: true, imports: [FormsModule, HttpClientModule], template: `

Spam Classifier

Classify
Result: **{{ result }}**

```
  `,
  styles: [`
    .container{max-width:480px;margin:40px auto;padding:20px;border:1px solid #ddd;backgrc
    textarea{width:100%;font-size:1rem}
    button{margin-top:10px;padding:8px 16px}
  `]
```

})
export class AppComponent {
/* component logic unchanged */
}

```
2. **…or generate separate files** (if you prefer the old layout)
```

```
Angular still lets you externalise the template and styles:

```bash
ng generate component home --standalone --flat --inline-template=false --inline-style=fal
```

This produces `home.component.html` and `home.component.css`.
You could delete the original inline content and move everything into these new files.

3. **Why the documentation you saw still shows** *.html*
   Many blog posts and tutorials were written for Angular ≤16. The CLI's switch to standalone components removed NgModule boilerplate and defaulted to inline templates for new projects, hence the discrepancy you observed.

Once you decide on inline versus external templates, continue with the earlier steps (add FormsModule/HttpClientModule imports, write the classify() method, build, deploy, etc.).

⁂

Why `app.module.ts` Is Gone — and How to Proceed

Starting with **Angular 17**, the CLI defaults to the **stand-alone application model**.
In a stand-alone app:

- Components declare their own dependencies via the `imports` array inside the `@Component` decorator.
- The bootstrap process is handled in `main.ts` with `bootstrapApplication()` instead of a root `NgModule`.
- Therefore `app.module.ts` **is not generated** unless you explicitly ask for a module-based ("legacy") structure.

You now have two choices:

1. Stay in Stand-Alone Mode (Recommended)

Nothing is missing—just keep working with the files you already have:

```
src/app/
│
├─ app.component.ts   ← inline template + styles
├─ app.config.ts      ← providesRouter(), HttpClient, etc.
└─ main.ts            ← calls bootstrapApplication()
```

To add libraries (Forms, HTTP) or other components, import them directly in `app.component.ts` (or any other standalone component):

```
@Component({
  selector: 'app-root',
  standalone: true,
```
```

```
  imports: [FormsModule, HttpClientModule],
  template: ` ... `,
})
export class AppComponent { /* … */ }
```

Your small spam-classifier UI works perfectly in this setup; the earlier code sample merely moves the HTML and CSS inline instead of into separate files.

2. Switch Back to Module-Based Structure

If you prefer the classic `AppModule`/NgModule workflow, you have two paths:

1. **Create a fresh module-based project and copy code**

   ```
   ng new spam-ui --no-standalone --routing=false --style=css
   ```

   This generates the familiar `app.module.ts`, `app.component.html`, etc.
   Move your component logic and HTML into the new files.

2. **Add a module to the existing project**
   Generate a root module and migrate bootstrapping:

   ```
   ng generate module app --flat    # creates src/app/app.module.ts
   ```

   In `app.module.ts`:

   ```
   import { BrowserModule } from '@angular/platform-browser';
   import { NgModule } from '@angular/core';
   import { FormsModule } from '@angular/forms';
   import { HttpClientModule } from '@angular/common/http';
   import { AppComponent } from './app.component';

   @NgModule({
     declarations: [AppComponent],
     imports: [BrowserModule, FormsModule, HttpClientModule],
     bootstrap: [AppComponent],
   })
   export class AppModule {}
   ```

   Then replace `bootstrapApplication(AppComponent)` in `main.ts` with:

   ```
   platformBrowserDynamic().bootstrapModule(AppModule)
     .catch(err => console.error(err));
   ```

Either approach restores the module-based pattern, but the Angular team encourages new apps to stay standalone unless you have a specific need for `NgModule`s.

Which Should You Use?

For a *tiny* single-page front-end that only calls `/predict`, **stand-alone mode keeps the project lighter and simpler**—no extra files, no boilerplate module. You simply:

1. Import `FormsModule` and `HttpClientModule` in `app.component.ts`.
2. Keep the inline template/styles or generate external files with

   ```
   ng generate component home --standalone --flat --inline-template=false --inline-style
   ```

3. Build and deploy as previously outlined.

If later you expand the UI into a larger multi-feature app, you can still add modules or convert the project with Angular's official migration tooling.

⁂

1. https://angular.dev/cli/new
2. https://v17.angular.io/tutorial/tour-of-heroes/toh-pt0
3. https://stackoverflow.com/questions/47155209/create-angular-cli-app-without-default-packages-for-tooling
4. https://developer.mozilla.org/en-US/docs/Learn_web_development/Core/Frameworks_libraries/Angular_getting_started
5. https://github.com/angular/angular-cli/issues/5431
6. https://nx.dev/blog/architecting-angular-applications
7. https://angular.io/cli/generate
8. https://angular.dev/tools/libraries/angular-package-format
9. https://www.angulararchitects.io/blog/guide-for-ssr/
10. https://stackoverflow.com/questions/77454741/why-doesnt-app-module-exist-in-angular-17
11. https://github.com/angular/angular/issues/52751
12. https://www.linkedin.com/pulse/how-fix-missing-appmodulets-file-latest-version-angular-sofia-nayak-er0df
13. https://v17.angular.io/tutorial/tour-of-heroes/toh-pt0
14. https://www.geeksforgeeks.org/angular-js/create-project-with-app-module-in-angular-17/
15. https://stackoverflow.com/questions/77526846/why-app-module-ts-file-is-not-getting-created-whenever-i-create-a-new-angular-ap/77527319
16. https://angular.dev/cli/new
17. https://www.youtube.com/watch?v=Lkz3rYqck08
18. https://angular.dev/reference/migrations/standalone